To appear in *Proc., COMPASS ’97*

Tools for Formal Specification, Verification, and Validation of Requirements

Constance Heitmeyer, James Kirby, and Bruce Labaw[*]
Center for High Assurance Computer Systems (Code 5546)
Naval Research Laboratory, Washington, DC 20375
{heitmeyer, kirby, labaw}@itd.nrl.navy.mil

Abstract

Although formal methods for developing computer systems have been available for more than a decade, few have had significant impact in practice. A major barrier to their use is that software developers find formal methods difficult to understand and apply. One exception is a formal method called SCR for specifying computer system requirements which, due to its easy to use tabular notation and its demonstrated scalability, has already achieved some success in industry. Recently, a set of software tools, including a specification editor, a consistency checker, a simulator, and a verifier, has been developed to support the SCR method [9, 11, 5]. This paper describes recent enhancements to the SCR tools: a new dependency graph browser which displays the dependencies among the variables in the specification, an improved consistency checker which produces detailed feedback about detected errors, and an assertion checker which checks application properties during simulation. To illustrate the tool enhancements, a simple automobile cruise control system is presented and analyzed.

1. Introduction

Although formal methods for developing computer systems have been available for more than a decade, few of these methods have had significant impact in the development of practical systems. A major impediment to the use of formal methods in industrial software development is the widespread view that the methods are impractical. Not only do developers regard most formal methods as difficult to understand and apply. In addition, they have serious doubts about the scalability and cost-effectiveness of the methods.

A promising approach to overcoming these problems is to hide the logic languages associated with most formal methods and to adopt a notation, such as a graphical or tabular notation, that developers find easier to user. Specifications in the more “user-friendly" notation can be translated automatically to a form more amenable to formal analysis. Moreover, to scale effectively, a formal method must be supported by powerful, easy to use tools. To the extent feasible, the tools should detect software errors automatically and provide easy to understand feedback useful in tracing the cause of an error.

By providing a “user-friendly" tabular notation with demonstrated scalability, a formal method called SCR for specifying the requirements of computer systems has already achieved some success in practice. Since SCR’s introduction more than a decade ago [13, 1], many industrial organizations, including Lockheed, Grumman, and Ontario Hydro, have used the SCR method to specify requirements. To support the SCR method, we have recently developed a set of integrated software tools [9, 11, 5] to specify and analyze system and software requirements. The tools include a *specification editor* for creating and modifying a requirements specification, a *simulator* for symbolically executing the specification, a *consistency checker* which checks the specification for well-formedness (e.g., syntax and type correctness, no missing cases, no circular definitions, and no unwanted nondeterminism), and a *verifier* for analyzing the specification for critical application properties.

To place SCR specifications in perspective, this paper first compares an SCR requirements specification with two other specifications, an abstract model useful in verification and a specification using the commercially available product STATEMATE. It then describes the current status of the SCR tools, including three major enhancements added since the publication of [10, 9]. These are a new dependency graph browser which displays the dependencies among the different variables in the specification, an improved consistency checker which produces examples of missing cases and nondeterminism when either a coverage or disjointness error is detected, and an assertion checker which tests various application properties during simulation. We also show how our tools support DURATION, a language feature originally proposed by van Schouwen [25] to represent time in SCR specifications. To illustrate the SCR method and our tools, a requirements specification of a simple automobile cruise control system is presented and analyzed. Finally, we present some lessons learned in experimental use of our tools in industrial applications.

[*]This work was supported by ONR and SPAWAR.

Report Documentation Page	*Form Approved OMB No. 0704-0188*

1. REPORT DATE **1997**	2. REPORT TYPE	3. DATES COVERED **00-00-1997 to 00-00-1997**

4. TITLE AND SUBTITLE **Tools for Formal Specification, Verification, and Validation of Requirements**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**13**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

2. SCR Method: An Overview

2.1. SCR Specifications

A recent article by Shaw [22] presents and discusses a number of different specifications of an automobile cruise control system. Each of these specifications is constructed to satisfy different objectives. For example, Atlee and Gannon use a logic language to specify the different “modes" of the cruise control system [4]. Their logic language specification is then fed to a model checker that analyzes the specification for violations of selected properties. Another specification of the cruise control system by Smith and Gerhart is represented using the graphical notation of STATEMATE and is described by the authors as a “design exercise" [23]. We refer to the former specification as an *abstract model* and the latter as the *STATEMATE specification*.

One difference between the abstract model, the STATEMATE specification, and an SCR specification of the cruise control system lies in the notation. The abstract model is expressed in a logic language, the STATEMATE specification in a graphical notation, and the SCR specification in a tabular format. Another difference is the target audience. The abstract model is designed to be processed by a computer, whereas both the SCR specification and the STATEMATE specification are engineering documents, designed to be read by software developers. The three specifications also differ in a third respect—namely, in the specific information each contains about the required system behavior.

The objective of the SCR specification is to describe the externally visible behavior of the Cruise Control System. To achieve this, the specification must describe the required relation REQ between the *monitored variables*, which represent quantities in the environment that the system monitors, and the *controlled variables*, which represent environmental quantities that the system controls. In the cruise control system, the position of the cruise control lever is an example of a monitored variable; the position of the throttle is an example of a controlled variable. The REQ relation between the monitored and controlled variables is one of the four relations of the Parnas-Madey Four Variable Model, a formal framework for describing the required behavior of a computer system [20].

Atlee and Gannon’s abstract model is used in verification and, as a result, omits many details of the required system behavior. For example, it does not describe the behavior of the throttle. Because the properties analyzed in [4] are independent of the throttle behavior and because a model used in verification should only include information needed to reason about selected properties, omitting information about the throttle is appropriate. In fact, eliminating irrelevant information is especially important for model checking: without dramatic reductions in the size of the state space to be analyzed, model checking is infeasible.

The STATEMATE specification is in some respects more detailed and in other respects less detailed than the SCR specification. For example, it presents two views of the required behavior, the functional view and the behavioral view, and distinguishes control flows, data flows, and data stores. The SCR requirements specification omits this detail (as does the abstract model) because such detail is unneeded for describing externally visible behavior. It presents a single “view" of the required behavior and makes no distinction between control flow and data flow nor between data flows and data stores.

The SCR environmental variables (the monitored and controlled variables) are often more abstract than the variables selected when other methods are applied. For example, the SCR specification of cruise control uses the monitored variable `Speed` to denote the speed of the automobile. In contrast, the STATEMATE specification uses calculations based on rotations of the automobile’s drive shaft to represent the automobile’s speed. In other cases, the SCR environmental variables are less abstract than the variables selected using other methods. The SCR specification of cruise control is explicit about the relationship between the value of the monitored variable `Lever` and the position in which the driver holds the cruise control lever. In contrast, the STATEMATE specification abstracts from much that is directly observable by the driver.

In contrast to the abstract model, the SCR requirements specification is a repository for all of the information that developers will need to construct the software for the cruise control system. Hence, it is necessarily more detailed and less abstract than a model useful in verification. At the same time, an SCR specification contains less information than, say, a software design document, since its goal is to describe the blackbox behavior of the system only.

The SCR requirements method provides detailed guidance on exactly what information belongs in a requirements document, a conceptual model of the system to be developed, and special language constructs to represent the system requirements. This detailed guidance, system model, and language constructs specialized for requirements specification are lacking in an approach based on STATEMATE because STATEMATE, a general-purpose method that can be applied throughout software development, is not customized for requirements specification.

Although the requirements specification for the Cruise Control System presented in this paper is close to a “real" requirements specification useful to software developers, three classes of information must be added for the specification to be complete: a description of the I/O devices the system uses to measure and compute the monitored and controlled quantities, the required timing and accuracy, and the constraints imposed on the system by physical laws and the environment (the relation NAT in the Four Variable Model).

2.2. SCR Requirements Model

To provide a precise and detailed semantics for the SCR method, we have developed the SCR requirements model, which represents a system as a finite state automaton and describes the monitored and controlled variables and other constructs that make up an SCR specification in terms of that automaton [12, 11]. To concisely describe the required relation between the monitored and controlled variables, our model uses four constructs—modes, terms, conditions, and events. A *mode class* is a partitioning of the system states. Each equivalence class in the partition is called a *system mode* (or simply *mode*). A *term* is any function of monitored variables, modes, or other terms. A *condition* is a predicate defined on a system state. An *event* occurs when the value of any system variable changes (a system variable is a monitored or controlled variable, a mode class, or a term). The notation "`@T(c) WHEN d`" denotes a *conditioned event,* which is defined by

$$\texttt{@T(c) WHEN d} \overset{\text{def}}{=} \neg c \wedge c' \wedge d,$$

where the unprimed condition c is evaluated in the "current" state, and the primed condition c' is evaluated in the "new" state. The environment may change a monitored quantity, causing an *input event*. In response, the system updates terms and mode classes and changes controlled quantities.

2.3. Cruise Control System

To illustrate the SCR constructs, this paper contains a specification of a real automobile cruise control system originally specified by Kirby [16]. To make the specification more understandable, our specification describes only a subset of the behavior in the original. A special feature of the Cruise Control System is that time is an important monitored quantity. For example, to cause the throttle to accelerate the automobile, the driver must hold the cruise control lever in the `const` positon for more than 1/2 second. Below, we show how the DURATION construct can be used to represent this and other timing behavior in the specification of the Cruise Control System.

The Cruise Control System monitors several quantities in the automobile's environment, such as the ignition switch, the position of the cruise control lever, the automobile's speed, and a service reset switch, and uses this information to control a throttle and to determine when a service light illuminating the message, "Major Service Required" is off, on, or flashing (i.e., intermittent). For example, if the ignition is on, the engine running, and the brake off, the driver can invoke cruise control by moving the cruise control lever to the `const` position. Once cruise control has been invoked, the system uses the automobile's actual speed to determine whether to set the throttle to accelerate or decelerate the automobile, or to maintain the current speed. The driver overrides cruise control by engaging the brake, resumes cruise control by moving the lever to `resume`, and exits cruise control by moving the lever to `off`. To determine when to illuminate the service light, the system computes the number of miles traveled since the last maintenance and illuminates the light intermittently when one threshold is reached and continuously when a higher threshold is reached.

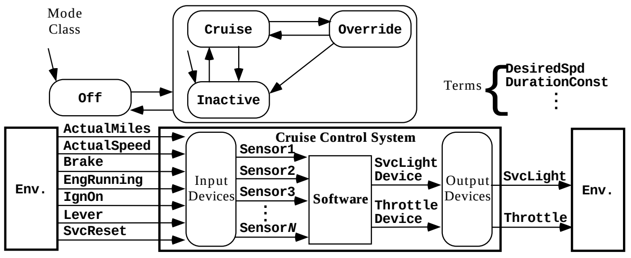
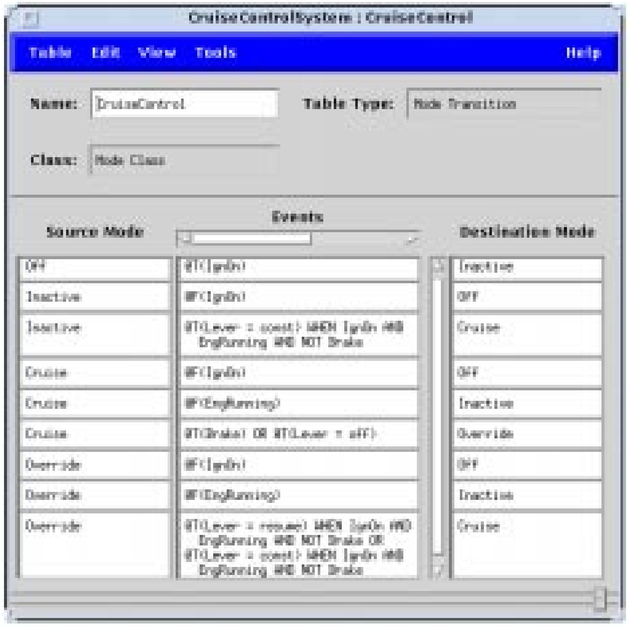


Figure 1. Cruise Control Specification.

Figure 1 shows how SCR constructs can be used to specify the requirements of the Cruise Control System. The monitored variables, `IgnOn`, `EngRunning`, `ActualMiles`, `ActualSpeed`, `Brake`, `Lever`, and `SvcReset`, represent the state of the automobile's ignition and engine (each is off or on), the readings of the odometer and speedometer, and the positions of the brake, cruise control lever, and service reset switch. Although it is not shown in Figure 1, the distinguished monitored variable `Time` is also required in this specification. The controlled variables, `Throttle` and `SvcLight`, represent the state of the throttle and the service light.

The specification contains one mode class and three terms. The mode class `CruiseControl` contains four modes, `Off`, `Inactive`, `Cruise`, and `Override`. At any given time, the system must be in one of these modes. Turning the ignition on causes the system to leave `Off` mode and enter `Inactive` mode, while turning the cruise control level to `const` when the brake is off and the engine running causes the system to enter `Cruise` mode. To override cruise control (i.e., enter `Override`), the driver turns the lever to `off` or applies the brake. The term `DesiredSpd` is set to the automobile's actual speed under certain conditions, e.g., if the driver turns the lever to `const` when the ignition is on, the engine running, and the brake off. The term `SvcMiles` contains the number of miles driven since the last maintenance; the term `MilesSvcReset` is used to compute `SvcMiles`.

The specification also includes several conditions and events. An example of a condition in the specification is "`DesiredSpd` $>$ `ActualSpeed`". Two examples of input events are "@T(`Lever`$=$`off`)" (the driver moves `Lever` from a position, such as `release`, to `off`) and "@F(`IgnOn`)" (the driver turns the ignition off), where

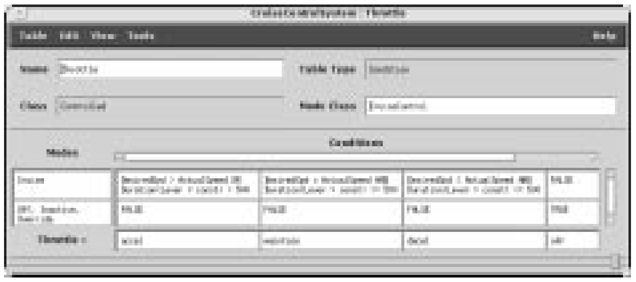


Name: CruiseControl   Table Type: Mode Transition

Class: Mode Class

Source Mode	Events	Destination Mode
OFF	@T(IgnOn)	Inactive
Inactive	@F(IgnOn)	OFF
Inactive	@T(Lever = const) WHEN IgnOn AND EngRunning AND NOT Brake	Cruise
Cruise	@F(IgnOn)	OFF
Cruise	@F(EngRunning)	Inactive
Cruise	@T(Brake) OR @T(Lever = off)	Override
Override	@F(IgnOn)	OFF
Override	@F(EngRunning)	Inactive
Override	@T(Lever = resume) WHEN IgnOn AND EngRunning AND NOT Brake OR @T(Lever = const) WHEN IgnOn AND EngRunning AND NOT Brake	Cruise

Table 1. Mode Table for `CruiseControl`**.**



Name Throttle   Table Type Condition

Class Controlled   Mode Class CruiseControl

Modes	Conditions			
Cruise	DesiredSpd > ActualSpeed OR Duration(Lever = const) > 500	DesiredSpd = ActualSpeed AND Duration(Lever = const) <= 500	DesiredSpd < ActualSpeed AND Duration(Lever = const) <= 500	FALSE
OFF, Inactive, Override	FALSE	FALSE	FALSE	TRUE
Throttle =	accel	maintain	decel	off

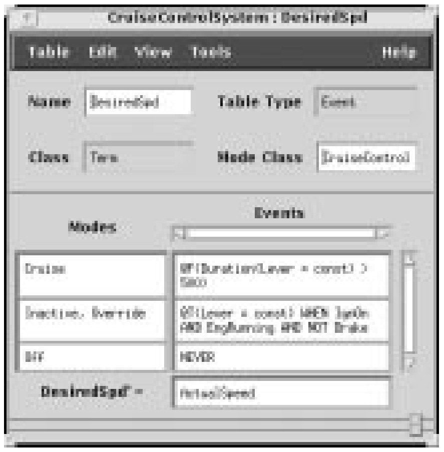
Table 2. Condition Table for `Throttle`**.**

"@T(A)" denotes the event of A becoming true, "@F(A)" denotes the event of A becoming false. An example of a conditioned event is "@T(`Lever`=`resume`) WHEN `IgnOn` $\wedge$ `EngRunning` $\wedge$ $\neg$`Brake`" (the driver moves `Lever` to `resume` when the ignition is on, the engine running, and the brake off).

2.4. SCR Tables

Among the tables in SCR specifications are condition, event, and mode transition tables. Each table defines a mathematical function. A condition table describes a controlled variable or term as a function of a mode and a *condition*; an event table describes a controlled variable or term as a function of a mode and an *event*. A mode transition table describes a mode as a function of a mode and an event.

Tables 1–4, each constructed with our toolset, are part of the system requirements specification for the Cruise Control System. Table 1 is a mode transition table describing the mode class `CruiseControl` as a function of the current mode and the monitored variables `Lever`, `IgnOn`, `EngRunning`, and `Brake`. The table defines all events that change the value of `CruiseControl`. For example, the third row states, "If `CruiseControl` is `Inactive`, `IgnOn` and `EngRunning` are true, `Brake` is false (i.e., off), and the driver moves `Lever` to `const`, then `CruiseControl` enters the mode `Cruise`." Events that do not change the value of the mode class are omitted from the table.



Name DesiredSpd   Table Type Event

Class Term   Mode Class CruiseControl

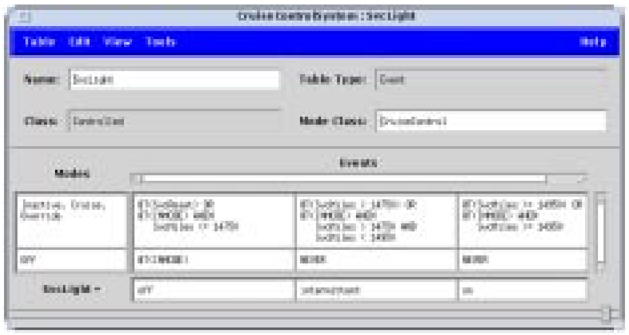
Modes	Events
Cruise	@F(Duration(Lever = const) > 500)
Inactive, Override	@T(Lever = const) WHEN IgnOn AND EngRunning AND NOT Brake
OFF	NEVER
DesiredSpd' =	ActualSpeed

Table 3. Event Table for `DesiredSpd`**.**

Table 2 is a condition table describing the controlled variable `Throttle` as a function of the current mode and the variables `ActualSpeed`, `DesiredSpd`, and `Lever`. It uses a new language feature supported by our tool called DURATION. This feature allows the specifier to define a predicate on the length of time the system has been in a given state. To illustrate how DURATION is used, we consider the condition, "DURATION(Lever = const) $>$ 500" in the first row of Table 2. This condition is *true* if the cruise control lever has been in `const` for more than 500 ms. Thus, if the lever enters the `const` position at time t and remains there for 600 ms, at time $t + 400$, the condition "DURATION(Lever = const) $>$ 500" is *false*; at time $t+550$, the condition is *true*.

Table 3 is an event table describing the term `DesiredSpd` as a function of the current mode and the variables `IgnOn`, `EngRunning`, `Brake`, and `Lever`. Like mode transition tables, event tables make explicit only those events that cause the variable defined by the table to change. Table 4 is also an event table. It describes the controlled variable `SvcLight` as a function of the current mode and the variables `SvcReset` and `SvcMiles`.

To illustrate how SCR tables can be transformed into functions, we present below the function that can be derived from Table 1 using our model [11]. This function describes the required behavior of the mode class `CruiseControl`. (To save space, we abbreviate `CruiseControl` as `CC` and `EngRunning` as `EngR`.)



Name: SvcLight Table Type: Event
Class: Controlled Mode Class: CruiseControl

Modes	Events		
Inactive, Cruise, Override	@T(SvcReset) OR @T(INMODE) WHEN SvcMiles < 1470	@T(SvcMiles >= 1470) OR @T(INMODE) WHEN SvcMiles >= 1470 AND SvcMiles < 1490	@T(SvcMiles >= 1490) OR @T(INMODE) WHEN SvcMiles >= 1490
Off	@T(INMODE)	NEVER	NEVER
SvcLight' =	off	intermittent	on

Table 4. Event Table for `SvcLight`.

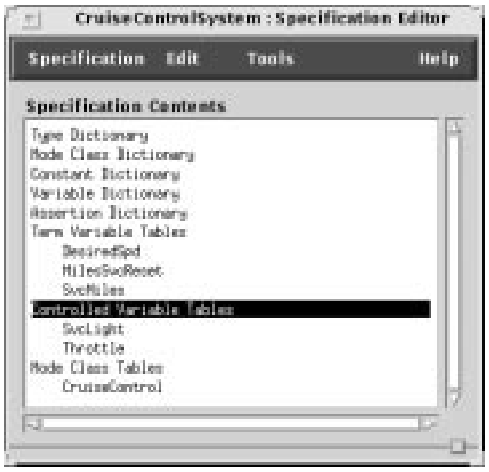
$$
\texttt{CC}' =
\begin{cases}
\texttt{Off} & \text{if } [\texttt{CC=Inactive} \wedge \texttt{@F(IgnOn)}] \vee \\
 & \quad [\texttt{CC=Cruise} \wedge \texttt{@F(IgnOn)}] \vee \\
 & \quad [\texttt{CC=Override} \wedge \texttt{@F(IgnOn)}] \\
\texttt{Inactive} & \text{if } [\texttt{CC=Off} \wedge \texttt{@T(IgnOn)}] \vee \\
 & \quad [\texttt{CC=Cruise} \wedge \texttt{@F(EngR)}] \vee \\
 & \quad [\texttt{CC=Override} \wedge \texttt{@F(EngR)}] \\
\texttt{Cruise} & \text{if } [\texttt{CC=Inactive} \wedge \texttt{@T(Lever=const)} \\
 & \quad \wedge \texttt{IgnOn} \wedge \texttt{EngR} \wedge \neg\texttt{Brake}] \vee \\
 & \quad [\texttt{CC=Override} \wedge [\texttt{@T(Lever=resume)} \\
 & \quad \vee \texttt{@T(Lever=const)}] \wedge \\
 & \quad \texttt{IgnOn} \wedge \texttt{EngR} \wedge \neg\texttt{Brake}] \\
\texttt{Override} & \text{if } \texttt{CC=Cruise} \wedge \\
 & \quad [\texttt{@T(Brake)} \vee \texttt{@T(Lever=Off)}] \\
\texttt{CC} & \text{otherwise}
\end{cases}
$$

The *dependencies set* D_r of a variable r is the set of variables that determine the value of the variable. Because variables in SCR specifications can depend on variables in both the current state and the new state, both the unprimed and primed versions of a variable can appear in a dependencies set. From Table 1, we can derive the dependencies set D_{CC} for the `CruiseControl` mode class:

{`CC`, `IgnOn`, `IgnOn'` ,`Brake`, `Brake'` , `Lever`, `Lever'` , `EngR`, `EngR'`}

The value `CruiseControl` is in the dependencies set D_{CC} because the new value of `CruiseControl` depends on its current value. Each dependencies set D_r is the union of two sets D_r^{old} and D_r^{new}, where D_r^{old} (the *old* dependencies set) is the set of variables on which variable r depends in the current state and D_r^{new} (the *new* dependencies set) is the set of variables on which r depends in the new state.

To avoid circular definitions, we require that the new dependencies sets define a partial order on the variables in an SCR specification [11, 12]. The monitored variables occur first in the partial order because they only depend on changes that occur in the environment (i.e., their dependencies sets are empty). The controlled variables occur last because they can depend on any of the other variables in the specification. The mode classes and terms appear in the partial order between the monitored variables and the controlled variables.


CruiseControlSystem : Specification Editor
Specification Edit Tools Help

Specification Contents
- Type Dictionary
- Mode Class Dictionary
- Constant Dictionary
- Variable Dictionary
- Assertion Dictionary
- Term Variable Tables
  - DesiredSpd
  - MilesSvcReset
  - SvcMiles
- Controlled Variable Tables
  - SvcLight
  - Throttle
- Mode Class Tables
  - CruiseControl

Figure 2. Contents of the Specification.

3. Current Status of the SCR Tools

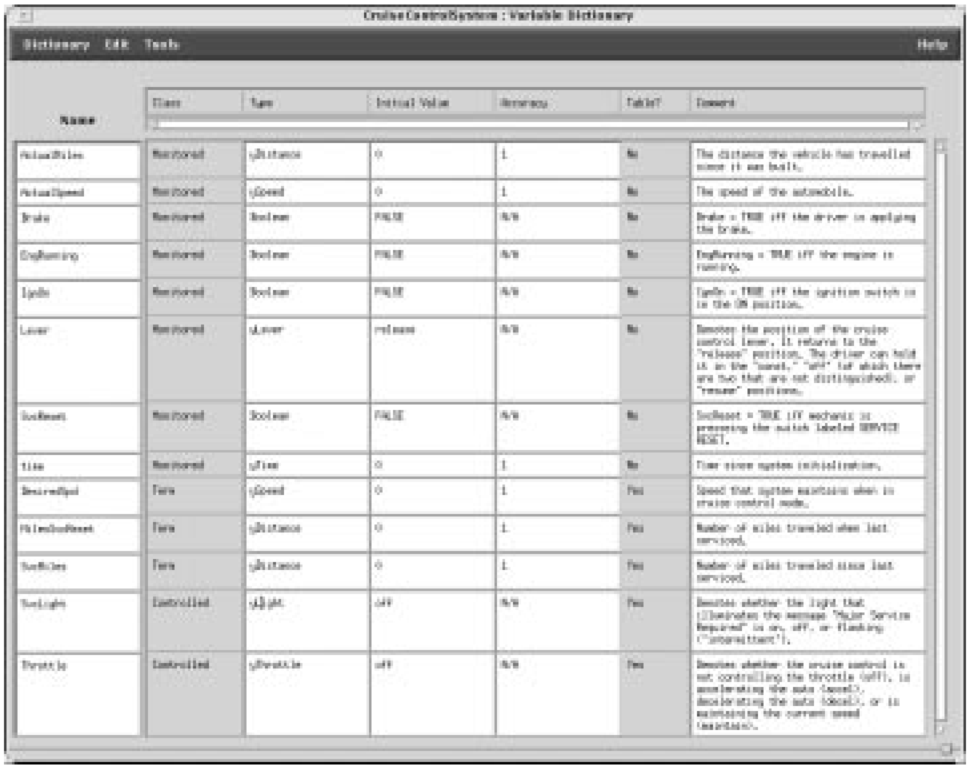
This section describes the current status of the five tools in our toolset: the specification editor, the dependency graph browser, the consistency checker, the simulator, and the verifier. The dependency graph browser and the verification capability are new. The specification editor, the consistency checker, and the simulator have recently undergone substantial improvements.

3.1. Specification Editor

To create, modify, or display a given requirements specification, the user invokes the specification editor. As illustrated in Figure 2, the editor lists the dictionaries and tables which make up the specification. The dictionaries define the static information in the specification, such as the names, values, and types of the variables; the user-defined types; etc. The tables are organized into three groups: tables defining terms, tables defining controlled variables, and mode transition tables.

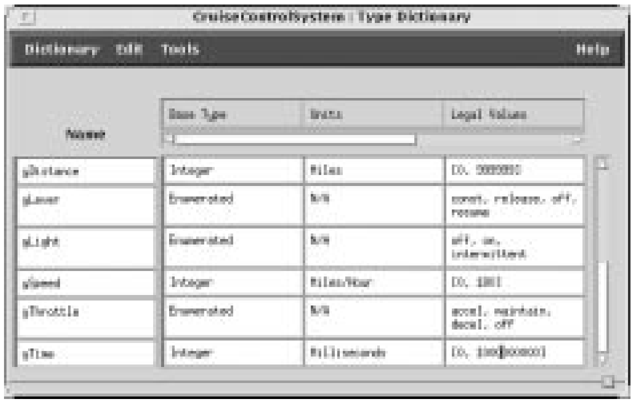
Each specification contains five dictionaries: the constant, type, mode class, variable, and assertion dictionaries. Figures 3–5 show the variable, type, and assertion dictionaries for the cruise control system. Time, which appears in both the variable dictionary in Figure 3 and the type dictionary in Figure 4, is represented as an non-negative integer with initial value zero. In each new state, time either stays the same or increases. In the Cruise Control specification, time is measured in milliseconds.

The *assertion dictionary*, a recent addition to the toolset, lists a set of application properties that the specifier can test via simulation, or verify using a model checker or mechanical theorem prover. The assertion dictionary shown in Figure 5 shows two kinds of properties: those that hold in



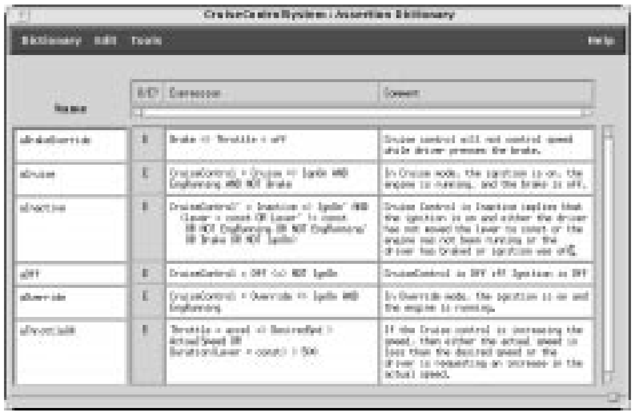
Name	Class	Type	Initial Value	Accuracy	Table?	Comment
ActualMiles	Monitored	yDistance	0	1	No	The distance the vehicle has travelled since it was built.
ActualSpeed	Monitored	ySpeed	0	1	No	The speed of the automobile.
Brake	Monitored	Boolean	FALSE	N/A	No	Brake = TRUE iff the driver is applying the brake.
EngRunning	Monitored	Boolean	FALSE	N/A	No	EngRunning = TRUE iff the engine is running.
IgnOn	Monitored	Boolean	FALSE	N/A	No	IgnOn = TRUE iff the ignition switch is in the ON position.
Lever	Monitored	yLever	release	N/A	No	Denotes the position of the cruise control lever. It returns to the "release" position. The driver can hold it in the "const," "off" (of which there are two that are not distinguished), or "resume" positions.
SvcReset	Monitored	Boolean	FALSE	N/A	No	SvcReset = TRUE iff mechanic is pressing the switch labeled SERVICE RESET.
time	Monitored	yTime	0	1	No	Time since system initialization.
DesiredSpd	Term	ySpeed	0	1	Yes	Speed that system maintains when in cruise control mode.
MilesSvcReset	Term	yDistance	0	1	Yes	Number of miles traveled when last serviced.
SvcMiles	Term	yDistance	0	1	Yes	Number of miles traveled since last serviced.
SvcLight	Controlled	yLight	off	N/A	Yes	Denotes whether the light that illuminates the message "Major Service Required" is on, off, or flashing ("intermittent").
Throttle	Controlled	yThrottle	off	N/A	Yes	Denotes whether the cruise control is not controlling the throttle (off), is accelerating the auto (accel), decelerating the auto (decel), or is maintaining the current speed (maintain).

Figure 3. Variable Dictionary.



Name	Base Type	Units	Legal Values
yDistance	Integer	Miles	[0, 3000000]
yLever	Enumerated	N/A	const, release, off, resume
yLight	Enumerated	N/A	off, on, intermittent
ySpeed	Integer	Miles/Hour	[0, 180]
yThrottle	Enumerated	N/A	accel, maintain, decel, off
yTime	Integer	Milliseconds	[0, 1000000000]

Figure 4. Type Dictionary.



Name	D/E?	Expression	Comment
aBrakeOverride	E	Brake => Throttle = off	Cruise control will not control speed while driver presses the brake.
aCruise	E	CruiseControl = Cruise => IgnOn AND EngRunning AND NOT Brake	In Cruise mode, the ignition is on, the engine is running, and the brake is off.
aInactive	E	CruiseControl' = Inactive => IgnOn' AND (Lever = const OR Lever' != const OR NOT EngRunning OR NOT EngRunning' OR Brake OR NOT IgnOn)	Cruise Control in Inactive implies that the ignition is on and either the driver has not moved the lever to const or the engine has not been running or the driver has braked or ignition was off.
aOff	E	CruiseControl = Off <=> NOT IgnOn	CruiseControl is OFF iff Ignition is OFF
aOverride	E	CruiseControl = Override => IgnOn AND EngRunning	In Override mode, the ignition is on and the engine is running.
aThrottleOK	E	Throttle = accel => DesiredSpd > ActualSpeed OR DurationLever = const) > 500	If the Cruise control is increasing the speed, then either the actual speed is less than the desired speed or the driver is requesting an increase in the actual speed.

Figure 5. Assertion Dictionary.

every reachable state and those that hold in any pair of adjacent reachable states. To make it easy for users to formulate the properties, the properties are expressed in standard logic rather than some special logic, such as temporal logic. The first property `aBrakeOverride`, which states "`Brake` ⇒ `Throttle`=`off`," is one that must hold in every reachable state. The assertion `aInactive` refers to variables in two adjacent states. In the column labeled "D/E?" in Figure 5, "E" and "D" indicate whether checking of the associated assertion is enabled or disabled.

3.2. Dependency Graph Browser

One criticism of SCR requirements specifications is that, while they give detailed information about specific aspects of the required system behavior, developing intuition about how the different parts of the specification are related is difficult, especially for large specifications. To address this problem, we have developed a dependency graph browser that displays the dependencies among the variables in a given specification. Figure 6 contains a graph showing the dependencies among the variables in the Cruise Control specification. Our tool constructed the graph automatically from the

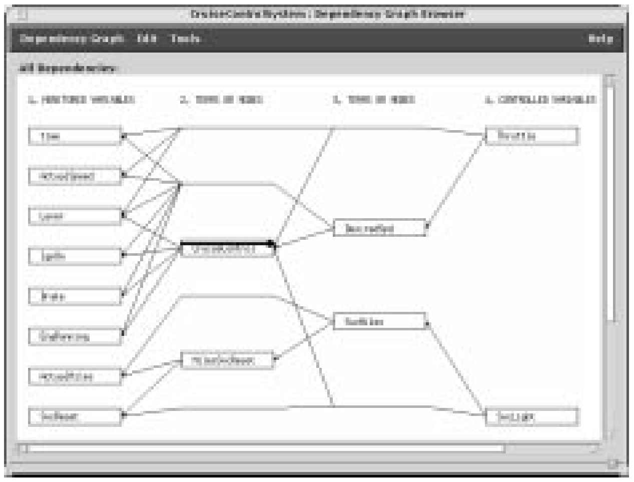


Figure 6. Dependency Graph.

requirements specification. The monitored variables appear as the leftmost nodes in the graph, the controlled variables appear on the right, and the nodes representing mode classes and terms appear in the middle.

The dependency graph in Figure 6 provides important information about the Cruise Control specification. For example, it shows that the mode class `CruiseControl` is defined in terms of the monitored variables `IgnOn`, `EngRunning`, `Brake`, and `Lever` as well as its previous value. Review of the graph also shows that the two controlled variables `Throttle` and `SvcLight` both depend on the mode class `CruiseControl`.

The dependency graph can reveal the presence of errors in the dependencies relation. Two kinds of errors are possible: circular dependencies and variables with incomplete definitions. Circular dependencies can only occur among variables in the new state. (The SCR semantics allow a variable to depend on *any* variable in the current state [11].) A variable has an incomplete definition when it does not lie on some path in the graph that includes both a controlled variable and a monitored variable. Two kinds of incompleteness can occur: a variable is not connected to a controlled variable (called a *right orphan*) or a variable which is not a monitored variable has a null dependencies set (called a *left orphan*). Figure 7 shows a graph with a cyclic dependency, and Figure 8 shows a dependency graph with a right orphan. In Figure 7, the thick arrow (between `MilesSvcReset` to `SvcLight`) indicates a circular definition involving the controlled variable `SvcLight` and the terms `SvcMiles`, and `MilesSvcReset`. In Figure 8, the node labeled `DesiredSpd` is a right orphan because it is a term on which no controlled variable depends.

Given a set of variables RF, the set of monitored variables $IR \subset RF$, the set of controlled variables $OR \subset RF$, and the dependencies sets D_r for each r in RF, we can construct the dependency graph for the variables in RF.

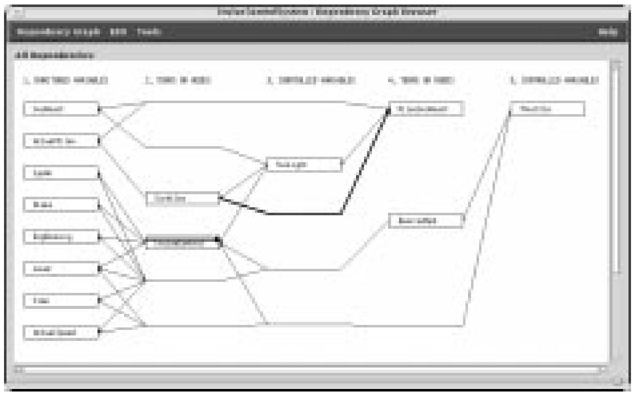


Figure 7. Graph with a Cycle.

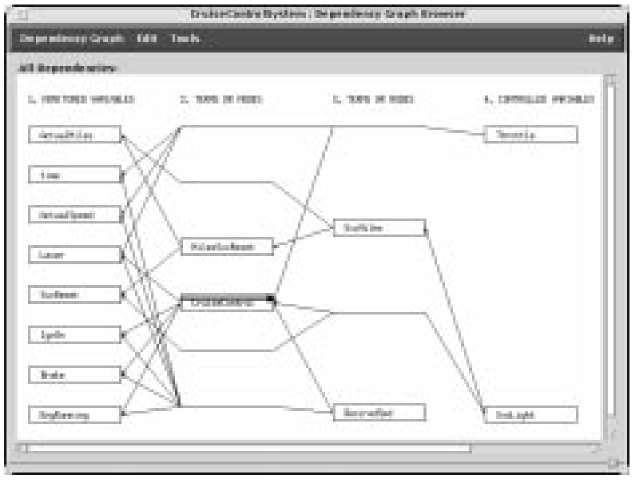


Figure 8. Graph with a Right Orphan.

Assuming that there are n levels in the dependency graph with the monitored variables at level 1 and the controlled variables at level n, then the sets of variables at each level, B_1, B_2, ..., B_n, can be computed recursively as follows:

Step 1:

- Let $B = RF - IR$ and $B_n = OR$.

For **Step** k, $k = 2, 3, \ldots, n-1$, compute B_{n-1}, B_{n-2}, ..., B_2 as follows:

- Remove all elements of B_{n-k+2} from B.
- If $B = \emptyset$, then go to Step n.
- Otherwise, $B_{n-k+1} = \{r \in B \mid \forall\, r' \in B : r \notin D_{r'}\}$.

Step n:

- $B_1 = IR$.

By using the dependencies sets D_r which describe all dependencies, the above algorithm produces the B_i's for the graph showing all dependencies. By using the new dependencies sets D_r^{new}, the above algorithm produces the B_i's for the

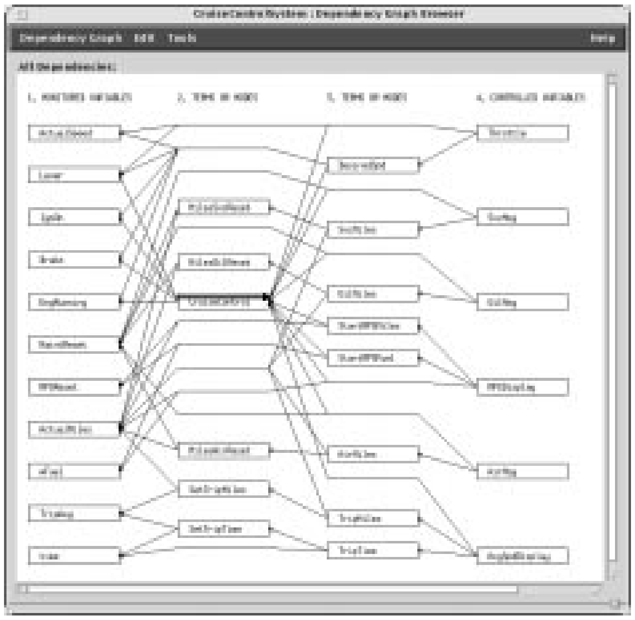


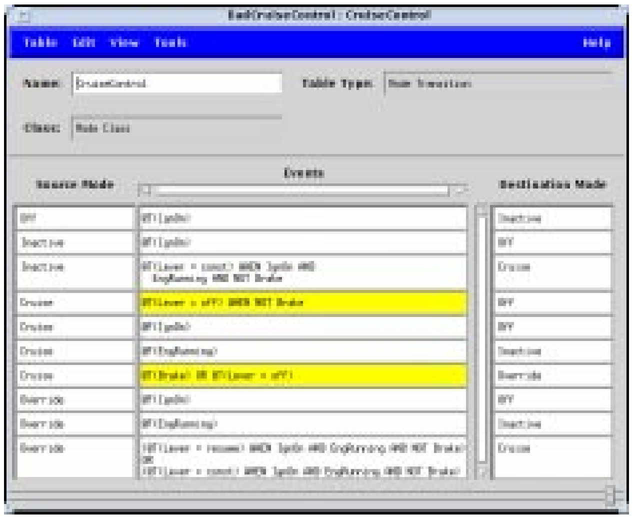
Figure 9. Dependency Graph of Complete Cruise Control.

graph showing the variable dependencies in the new state. A slight modification of the above algorithm is required in the presence of cycles.

For large specifications, the complete dependency graph is too big to fit on the user's display. Fortunately, a user typically wants to study only a small subset of the dependency graph. To display a subgraph, the user first finds the portion of the graph of interest and uses the mouse to select the variables of interest. Then, the user can display the subgraph containing all variables that each of the selected variables depends on, or alternatively all variables that depend on the selected variables. Figure 9 shows the dependency graph of the larger Cruise Control System originally specified by Kirby. Figure 6 shows the simple version of this system specified in this paper, which has fewer variables. To display this subset, the user selected the two controlled variables `Throttle` and `SvcLight` and then extracted the subgraph containing all variables on which these two variables depend.

3.3. Consistency Checker

Our consistency checker [11] verifies application-independent properties derived from our requirements model. These checks determine whether the specifications are well-formed. Among the errors the consistency checker detects are syntax and type errors, instances of incompleteness in the variable definitions and dictionaries, missing initial values, unreachable modes, and circular definitions



Name: CruiseControl
Table Type: Mode Transition
Class: Mode Class

Source Mode	Events	Destination Mode
Off	@T(IgnOn)	Inactive
Inactive	@F(IgnOn)	Off
Inactive	@T(Lever = const) WHEN IgnOn AND EngRunning AND NOT Brake	Cruise
Cruise	@T(Lever = off) WHEN NOT Brake	Off
Cruise	@F(IgnOn)	Off
Cruise	@F(EngRunning)	Inactive
Cruise	@T(Brake) OR @T(Lever = off)	Override
Override	@F(IgnOn)	Off
Override	@F(EngRunning)	Inactive
Override	(@T(Lever = resume) WHEN IgnOn AND EngRunning AND NOT Brake) OR (@T(Lever = const) WHEN IgnOn AND EngRunning AND NOT Brake)	Cruise

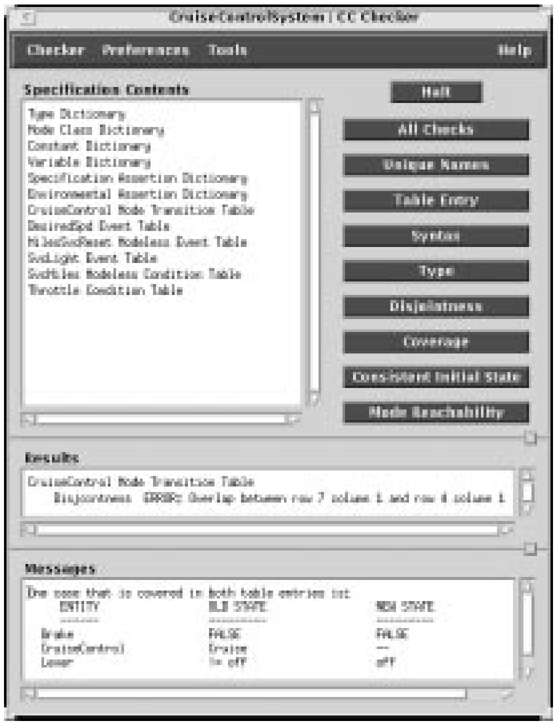
Figure 10. Disjointness Error.

(such as the one shown in Figure 7). The tool also checks for missing cases (called *Coverage* errors) and nondeterminism (called *Disjointness* errors).

To check for Disjointness and Coverage, the consistency checker determines whether a given logical expression is a tautology [11]. For example, to check two conditions c_1 and c_2 in a row of a condition table for Disjointness, the consistency checker evaluates the logical expression $c_1 \wedge c_2 =$ *false*. To check conditions $c_1, c_2, \ldots, c_n$ in a row of a condition table for Coverage, the tool evaluates the logical expression $\neg[c_1 \vee c_2 \vee \ldots \vee c_n] =$ *false*. To determine whether these logical expressions are tautologies, our tool applies a tableaux-based decision procedure that encodes the algorithm in [24].

When our consistency checker detects Coverage and Disjointness errors, it provides detailed feedback. The tool identifies the location of the error (e.g., the specific table entry or entries) and also provides an example of a system state or two adjacent system states containing the error. This detailed feedback significantly facilitates user correction of errors.

To illustrate the tool's handling of a disjointness error, we have modified Table 1 to include between rows 3 and 4 a new row stating, "If in `Cruise` mode the driver moves `Lever` to `off` when `Brake` is off, then the system enters mode `Off`." (See Figure 10.) To check the modified table for disjointness, we invoked the consistency checker. The Results Box in Figure 11 reveals a disjointness error. Double clicking on the line `Disjointness ERROR...` displays the modified table with the pair of entries that overlap highlighted. (See Figure 10.) This also causes a specific case of overlap to appear in the Messages Box of the Consistency Checker (see the bottom of Figure 11). This message states that any pair of adjacent states satisfying $\texttt{Lever} \neq \texttt{off} \wedge \texttt{Lever}' = \texttt{off} \wedge \neg\texttt{Brake} \wedge \texttt{CruiseControl} = \texttt{Cruise}$



Specification Contents

```
Type Dictionary
Mode Class Dictionary
Constant Dictionary
Variable Dictionary
Specification Assertion Dictionary
Environmental Assertion Dictionary
CruiseControl Mode Transition Table
DesiredSpd Event Table
MilesSvcReset Modeless Event Table
SvcLight Event Table
SvcMiles Modeless Condition Table
Throttle Condition Table
```

Results

```
CruiseControl Mode Transition Table
   Disjointness  ERROR: Overlap between row 7 column 1 and row 4 column 1
```

Messages

```
One case that is covered in both table entries is:
    ENTITY                 OLD STATE              NEW STATE
    ------                 ---------              ---------
  Brake                    FALSE                  FALSE
  CruiseControl            Cruise                 --
  Lever                    != off                 off
```

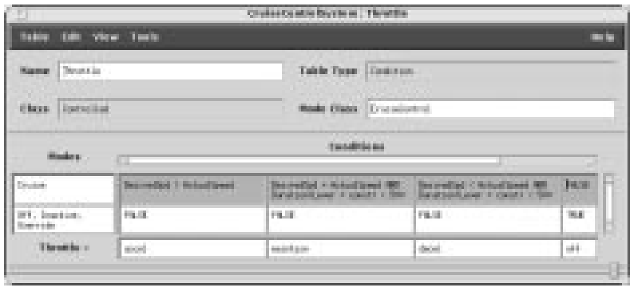
Figure 11. Feedback for Disjointness Error.

also satisfies both highlighted entries.

To check the two entries for Disjointness, the tautology checker evaluated the expression, [@T(`Lever=off`) $\wedge$ ¬`Brake`] $\wedge$ [@T(`Brake`) $\vee$ @T(`Lever=off`)] = *false*. The first disjunct in this expression, @T(`Lever=off`) $\wedge$ ¬`Brake` $\wedge$ @T(`Brake`) evaluates to *false* due to the assumption in our requirements model that only one monitored variable can change at each state transition [11]. The second disjunct, @T(`Lever=off`) $\wedge$ ¬`Brake` $\wedge$ @T(`Lever=off`), can be simplified to @T(`Lever=off`) $\wedge$ ¬`Brake`. Because this expression does not equal *false*, the statement is not a tautology. This expression provides the counterexample shown in the Results Box in Figure 11.

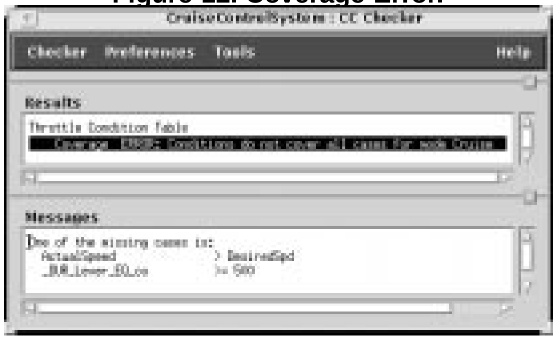
To illustrate the tool's handling of a coverage error, we modified the first row of Table 2 (see Figure 12). We then applied the consistency checker, which detected a coverage error. Double clicking on the line `Coverage ERROR...` displays the modified table with the row containing the relevant entries highlighted. This also causes an example of a missing case to appear in the Messages Box of the Consistency Checker (see the bottom of Figure 13): the table does not define the required behavior of Throttle for a system state satisfying `ActualSpeed` $>$ `DesiredSpd` $\wedge$ `DurationConst` $\geq$ 500.

A Coverage error occurs when the tautology checker processes a formula that does not evaluate to *false*. To find a counterexample, we can express the formula in Disjunctive Normal Form and then search for the first conjunct that does not evaluate to *false*. The tool presents this conjunct to the user as a counterexample.



Cruise	DesiredSpd ≤ ActualSpeed	[illegible]	[illegible]	[illegible]
Off, Inactive, Override	FALSE	FALSE	FALSE	TRUE
Throttle =	accel	maintain	decel	off

Figure 12. Coverage Error.


CruiseControlSystem : CC Checker
Checker Preferences Tools Help

Results

```
Throttle Condition Table
   Coverage  ERROR: Conditions do not cover all cases for mode Cruise
```

Messages

```
One of the missing cases is:
  ActualSpeed              > DesiredSpd
  _DUR_Lever_EQ_co         >= 500
```

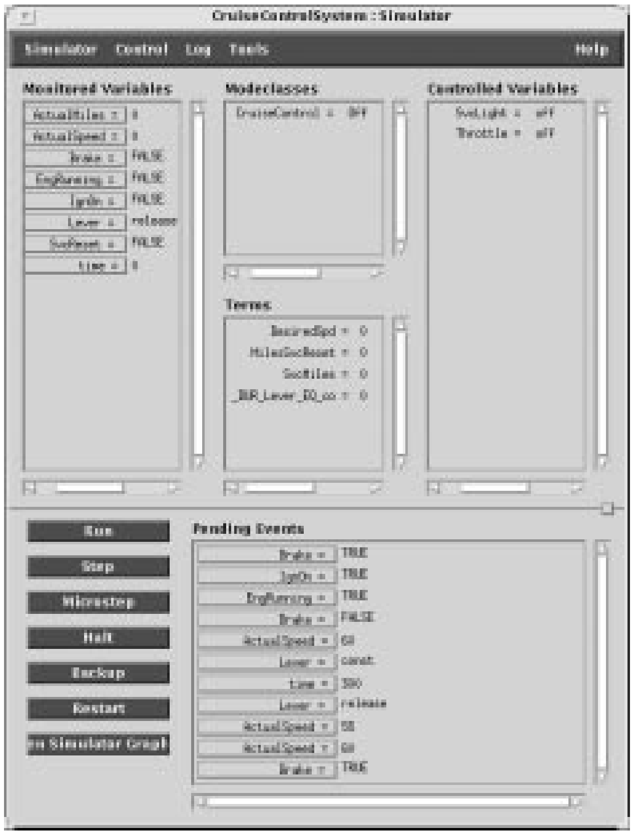
Figure 13. Feedback for Coverage Error.

To evaluate the first row of the table in Figure 12 for coverage, the tautology checker evaluated the expression[1] [`DesiredSpd` $\leq$ `ActualSpd`] $\wedge$ [(`DesiredSpd` $\neq$ `ActualSpd`) $\vee$ (`DurationConst` $\geq$ 500)] $\wedge$ [(`DesiredSpd` $\geq$ `ActualSpd`) $\vee$ (`DurationConst` $\geq$ 500)] $\wedge$ *true*. The first conjunct of the expression in Disjunctive Normal Form is [(`DesiredSpd` $\leq$ `ActualSpd`) $\wedge$ (`DesiredSpd` $\neq$ `ActualSpd`) $\wedge$ (`DesiredSpd` $\geq$ `ActualSpd`) $\wedge$ *true*), which reduces to *false*. The second conjunct is [(`DesiredSpd` $\leq$ `ActualSpd`) $\wedge$ (`DesiredSpd` $\neq$ `ActualSpd`) $\wedge$ (`DurationConst` $\geq$ 500) $\wedge$ *true*], which simplifies to (`DurationConst` $\geq$ 500) $\wedge$ (`DesiredSpd` $<$ `ActualSpd`). The tool presents this simple form to the user as a counterexample (see Figure 13).

3.4. Assertion Checking during Simulation

The user can validate the specification by executing the simulator and analyzing the result to ensure that the specification captures the intended behavior. In the new version of the simulator, the user can also define several properties believed to be true of the required system behavior and, us-

[1]Although our tool represents the condition DURATION(`Level=const`) $\geq$ 500 as _DUR_Lever_EQ_co $\geq$ 500, in the formulas we represent this expression as `DurationConst` $\geq$ 500. We represent other conditions containing DURATION similarly.



Monitored Variables

ActualMiles =	0
ActualSpeed =	0
Brake =	FALSE
EngRunning =	FALSE
IgnOn =	FALSE
Lever =	release
SvcReset =	FALSE
time =	0

Modeclasses

CruiseControl = Off

Terms

DesiredSpd = 0
MilesSvcReset = 0
SvcMiles = 0
_DUR_Lever_EQ_co = 0

Controlled Variables

SvcLight = off
Throttle = off

Pending Events

Brake =	TRUE
IgnOn =	TRUE
EngRunning =	TRUE
Brake =	FALSE
ActualSpeed =	60
Lever =	const
time =	300
Lever =	release
ActualSpeed =	55
ActualSpeed =	60
Brake =	TRUE

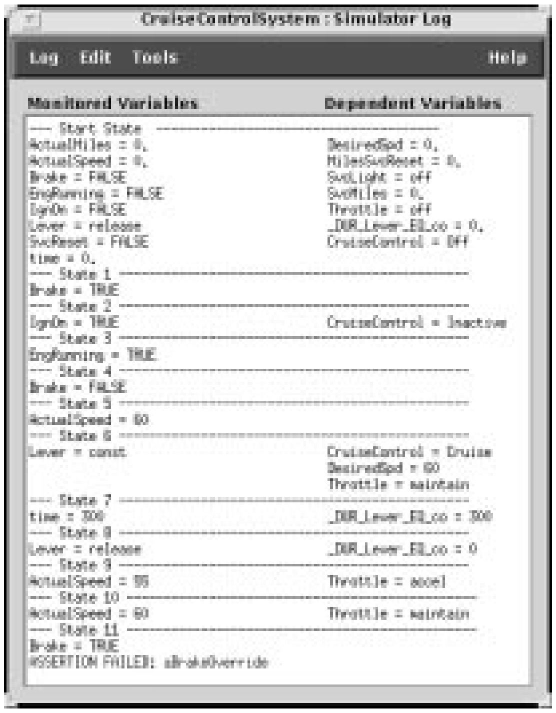
Figure 14. Simulator Display.

ing simulation, execute a series of scenarios to determine whether any violate the properties.

The user begins by invoking the simulator and then stepping through a scenario, a sequence of input events. To compute each new state from an input event and the current state, the simulator applies the transform (i.e., next-state) function of our requirements model. As each new state is computed, the Simulator window is updated to reflect the new state. The simulator also supports a second window, called the Log. The Log, which shows the state history, displays the initial state in full. For each subsequent state, it lists the input event that caused the transition along with each dependent variable (mode class, term, or controlled variable) whose value has changed.

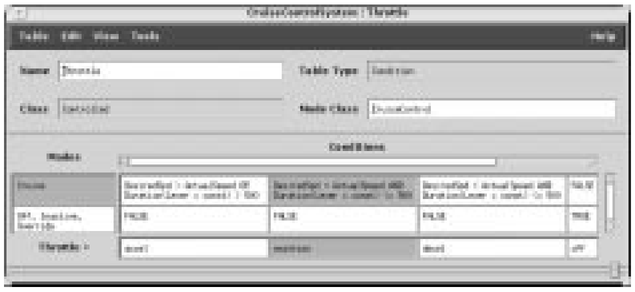
The scenario in Figure 15 demonstrates the behavior of the `Throttle` as defined by Table 2. When the ignition is turned on, the engine running, and the brake off, moving the cruise control lever to `const` causes the throttle to maintain the current speed of 60 mph (State 6). Once the driver releases the lever and the actual speed drops to 55 mph, the throttle accelerates the automobile (State 9).

To display the rule that caused a given dependent variable to change value, the user double clicks on the variable and its value in the Log. The simulator then displays the



```
--- Start State ----------------------------------------
ActualMiles = 0,                          DesiredSpd = 0,
ActualSpeed = 0,                          MilesSvcReset = 0,
Brake = FALSE                             SvcLight = off
EngRunning = FALSE                        SvcMiles = 0,
IgnOn = FALSE                             Throttle = off
Lever = release                           _DUR_Lever_EQ_co = 0,
SvcReset = FALSE                          CruiseControl = Off
time = 0,
--- State 1 --------------------------------------------
Brake = TRUE
--- State 2 --------------------------------------------
IgnOn = TRUE                              CruiseControl = Inactive
--- State 3 --------------------------------------------
EngRunning = TRUE
--- State 4 --------------------------------------------
Brake = FALSE
--- State 5 --------------------------------------------
ActualSpeed = 60
--- State 6 --------------------------------------------
Lever = const                             CruiseControl = Cruise
                                          DesiredSpd = 60
                                          Throttle = maintain
--- State 7 --------------------------------------------
time = 300                                _DUR_Lever_EQ_co = 300
--- State 8 --------------------------------------------
Lever = release                           _DUR_Lever_EQ_co = 0
--- State 9 --------------------------------------------
ActualSpeed = 55                          Throttle = accel
--- State 10 -------------------------------------------
ActualSpeed = 60                          Throttle = maintain
--- State 11 -------------------------------------------
Brake = TRUE
ASSERTION FAILED: sBrakeOverride
```

Figure 15. Log with a Violated Assertion.



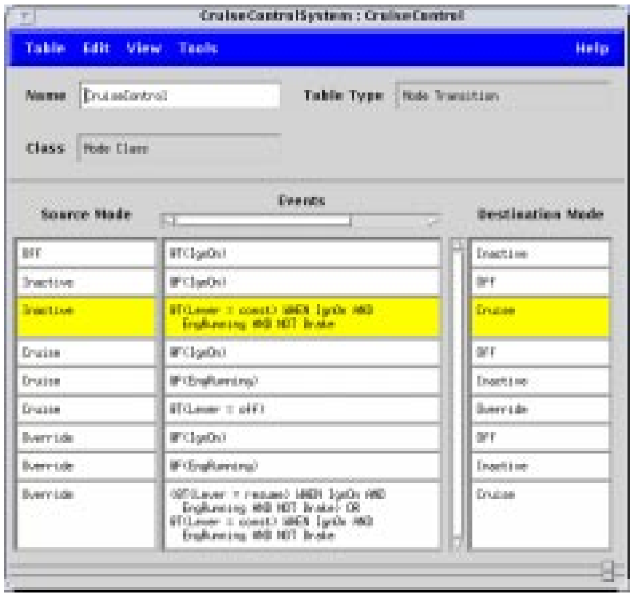
Name: Throttle
Table Type: Condition
Class: Controlled
Mode Class: CruiseControl

Modes	Conditions			
Cruise	DesiredSpd > ActualSpeed OR Duration(Lever = const) > 500	DesiredSpd = ActualSpeed AND Duration(Lever = const) <= 500	DesiredSpd < ActualSpeed AND Duration(Lever = const) <= 500	FALSE
Off, Inactive, Override	FALSE	FALSE	FALSE	TRUE
Throttle =	accel	maintain	decel	off

Figure 16. Table with Rule Highlighted.

table that defines the variable, highlighting the rule that caused the change. For example, double clicking on the expression "`Throttle = maintain`" in State 6 of the Log in Figure 15 causes the simulator to display the table, shown in Figure 16, that caused `Throttle` to change from `off` to `maintain`. As shown in Figure 16, the simulator has highlighted the rule that caused the change: "If the mode is `CruiseControl`, `DesiredSpd` equals `ActualSpeed`, and `DurationConst` is no more than 500 ms, then `Throttle` is `maintain`."

The bottom line of Figure 15 demonstrates the detection by the simulator of a violated assertion. The user simply clicks on the line in the Log reporting the violation to display the violated assertion. The tool then displays the assertion dictionary with the violated assertion highlighted (Figure 5, which omits the highlighting to improve readability). In this



Name: CruiseControl
Table Type: Mode Transition
Class: Mode Class

Source Mode	Events	Destination Mode
Off	@T(IgnOn)	Inactive
Inactive	@F(IgnOn)	Off
Inactive	@T(Lever = const) WHEN IgnOn AND EngRunning AND NOT Brake	Cruise
Cruise	@F(IgnOn)	Off
Cruise	@F(EngRunning)	Inactive
Cruise	@T(Lever = off)	Override
Override	@F(IgnOn)	Off
Override	@F(EngRunning)	Inactive
Override	(@T(Lever = resume) WHEN IgnOn AND EngRunning AND NOT Brake) OR @T(Lever = const) WHEN IgnOn AND EngRunning AND NOT Brake	Cruise

Figure 17. Mode Table with Rule Highlighted.

case, the violated assertion, `aBrakeOverride`, states that when the brake is applied, the throttle, which is controlled by the cruise control, should be off. Inspection of the Log (Figure 15) raises the question: Why didn't the throttle go off? Clicking on Throttle in State 10 of the Log (the last time Throttle changed) displays the table defining Throttle (see Figure 16). The table shows that `Throttle` = `off` only when the mode class `CruiseControl` ≠ `Cruise`. This raises the question: Why is `CruiseControl` = `Cruise` when the driver is pressing the brake (`Brake` = *true* in State 11)?

Clicking on `CruiseControl` in State 6 (the last time it changed) displays the mode transition table (Figure 17). We see that the system is still in mode `Cruise` while the brake is pressed because there is no transition out of mode `Cruise` when the driver presses the brake (denoted by the event @T(`Brake`)). The event in the sixth row of the mode transition table is @T(`Lever` = `off`). It should read @T(`Lever` = `off`) OR @T(`Brake`).

Assertion checking during simulation differs in important ways from model checking, a form of verification that checks all system states (or all pairs of adjacent system states) for violations. In contrast, assertion checking during simulation is a form of testing which only analyzes a small number of the possible states.

Assertion checking during simulation has an important advantage: it is much less expensive computationally than model checking. Although the complexity of model checking simple properties is linear with respect to the state space, the state space of SCR specifications, even small ones, is usually huge. For example, a simple analysis of the variable values and type information in the variable and type dictionaries (see Figures 3 and 4) shows that the simple cruise control system has over 10^{20} states. Because checking SCR specifications requires checking both the current state and the new state, the number of states to be analyzed exceeds 10^{40}! Hence, it makes sense to check assertions via simulation early in the development of the requirements specification to weed out errors; model checking is more cost-effective when the requirements specification is more mature. Assertion checking also requires much less effort. Before model checking can proceed, an abstract model with fewer states must be extracted from the specification. Generating such a model can be nontrivial.

3.5. Verifier

We recently integrated Spin [14], a tool that uses state exploration to verify properties of finite state machine models, into our toolset [5]. Our state-based requirements model provides a formal basis for using such a tool to verify requirements specifications. To do model checking, the user enters the property to be analyzed into the assertion dictionary and then invokes Spin from within the toolset to check the specification for the property. Because the large state space associated with most SCR specifications makes model checking infeasible, the user must provide Spin with an abstract model of the SCR specification. Reference [5] describes how our tool translates SCR specifications into Promela, the language of Spin, and the techniques we developed to generate abstract models from an SCR specification. To date, we have used Spin to verify two requirements specifications, one a small safety injection system [10] and the second a simple autopilot [6], for state invariant properties. These properties can involve any variables in the specifications—terms and controlled variables as well as mode classes and monitored variables.

4. Applying the Tools in Practice

Our tools, including two of the enhancements described above, have already proven valuable in ongoing experiments in which our group and colleagues in industry have applied the SCR method to practical applications. Recently, the SCR tools have been used extensively by engineers at Rockwell Collins Avionics & Communications to develop and to analyze an SCR specification of a reasonably large and complex avionics application. In developing the SCR specification, the engineers detected numerous errors, some in creating the SCR specification, others by running the simulator, and still others by applying consistency checking [18]. A software engineer at NORTEL has also used our tools to develop and analyze an SCR specification of a Steamer Boiler Controller Problem and also reports that the tools were helpful [26].

Our experience and the experience of colleagues at both Rockwell and NORTEL is that software tools are not only useful but essential for detecting errors in large specifications [18, 26]. Although manual inspections detect some errors, our tools find errors that manual inspections miss no matter how careful the developers are in preparing the specification. This is evidenced by the experience of an engineer at Rockwell, who reports that

> . . . even preliminary execution of the specification and completeness and consistency checking has found several errors in a specification that represented our best effort at producing a correct specification manually [18].

In the case of large specifications, detecting the cause of an error can be very difficult and time-consuming without counterexamples such as those described in Section 3.4. To determine the cause of an error, the developer could analyze the events and conditions highlighted by the tool, but such analysis is tedious and error-prone. Most often, the tool can find a counterexample much more quickly than a person can. Once the developer understands the error, he or she can look for a solution.

The dependency graph browser has been especially useful during the early stages in the development of an SCR requirements specification of a safety-critical Navy application. The basis for this effort is the specification produced by the Navy contractor. This specification, a combination of prose, diagrams, and formal statements of the required system properties, contains over 300 variables. Our understanding and that of the Navy manager of the dependencies among the many system variables has been aided enormously by the dependency graph generated by our tool. Without understanding how the variables are related, specifying the system behavior in SCR would be very hard.

Applying our techniques to practical systems has led us to improve the tools in other ways as well. For example, occasionally our consistency checker cannot determine in a short time whether a complex expression is a tautology. To prevent the tool from becoming mired in lengthy and complex analysis, we have installed a new tool feature which permits the user to set a maximum time for analysis of a given logical expression. This allows the checker to perform the easy analyses first. The more complex analyses can be postponed, or alternatively the user can study the expression that is causing a problem to determine why the analysis requires so much time.

Although assertion checking during simulation is available, our industrial colleagues have not yet used this feature. Once a more complete SCR specification of the Navy application is ready, we plan to use assertion checking during simulation to analyze the specification for the safety properties presented in the original contractor specification.

5. Related Work

The two techniques most closely related to ours are Tablewise [15] and the Requirements State Machine Language (RSML) and associated tools [17, 8]. Tablewise, a technique for processing decision tables, checks a table for both Disjointness (called "consistency") and Coverage (called "completeness"). It improves on earlier techniques based on decision tables by supporting nonboolean variables.

The primary goal in the RSML research is to develop techniques for producing safe systems. RSML, which was designed to describe real-time process control systems, uses a combination of the graphical Statecharts notation [7] and tables. A prototype tool has been developed for checking RSML specifications for "completeness" (i.e., every possible input event and the system's response to the event must be stated explicitly) and "consistency" (i.e., no input event can cause a transition to two different system states). The tool, which has been applied to large portions of the requirements specification of TCAS II, a collision avoidance system for commercial aircraft, detected errors not caught by an extensive manual review. Other tools are also being developed to analyze RSML specifications.

Another related system is the Prototype Verification System (PVS) [19], a specification and verification environment developed by SRI. PVS consists of a specification language, a type checker, and an interactive proof checker. The PVS specification language is based on a typed higher-order logic. The PVS prover performs a series of inference steps that can reduce a proof goal to simpler subgoals. These subgoals can be discharged automatically by the primitive proof steps of the prover. The primitive proof steps incorporate decision procedures for doing arithmetic, automatic rewriting, and BDD-based boolean simplification. Although some are attempting to use the language of PVS to produce requirements specifications, PVS is primarily designed to specify mathematical models and to prove theorems about those models using deductive reasoning supported by powerful decision procedures.

6. Conclusions

While the enhancements described in this paper are relatively straightforward, they have helped to make our tools significantly more useful for industrial-strength applications. Further enhancements to the tools are planned:

- Although the number of cases the consistency checker can handle has increased substantially in the period since its introduction, occasionally the checker encounters logical expressions that are too complex to analyze efficiently. We are investigating tools such as Omega [21] and PVS that can rewrite and simplify these expressions so they may be analyzed more efficiently.

- In another project, we have built an environment called TAME for specifying and proving properties about real-time systems on top of PVS [2, 3]. We are currently exploring the integration of TAME into our SCR toolset. Integration of TAME will allow the user of our tools to verify SCR specifications using a mechanical theorem prover.

Our hope is that our enhanced tools will be used to produce high-quality requirements specifications. These specifications should lead to systems that are more likely to perform as required and less likely to lead to accidents. The existence of high-quality requirements specifications should also lead to significant savings in software development costs.

Acknowledgments

We gratefully acknowledge the efforts of Carolyn Gasarch and Todd Grimm in developing the toolset software. We also thank Steve Miller of Rockwell and John van Schouwen of NORTEL for their extensive experiments with our tools and their helpful and detailed feedback. Many aspects of the tools have improved based on their comments. The second author extends his gratitude to Ramesh Bharadwaj for sharing his Latex secrets. We also appreciate the constructive comments of the anonymous referees.

References

[1] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore. Software requirements for the A-7E aircraft. Technical Report NRL-9194, Naval Research Lab., Wash., DC, 1992.

[2] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proc., Real-Time Applications Symposium*, 1996.

[3] M. Archer and C. Heitmeyer. TAME: A specialized specification and verification system for timed automata. In *Proc., Real-Time Systems Symposium (Work-in-Progress Session)*, 1996.

[4] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. Softw. Eng.*, 19(1):24–40, Jan. 1993.

[5] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. In *Proc., First ACM SIGPLAN Workshop on Automatic Analysis of Software*, 1997.

[6] R. W. Butler. An introduction to requirements capture using PVS: Specification of a simple autopilot. Technical Report NASA Technical Memorandum 110255, NASA Langley Research Center, Hampton VA, 1996.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8(3):231–274, June 1987.

[8] M. P. E. Heimdahl and N. Leveson. Completeness and consistency analysis of state-based requirements. In *Proc. of 17th Int'l Conf. on Softw. Eng. (ICSE '95)*, pages 3–14, Seattle, WA, Apr. 1995. ACM.

[9] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS '95)*, pages 109–122, Gaithersburg, MD, June 1995.

[10] C. Heitmeyer, B. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proc., International Symposium on Requirements Engineering*, Mar. 1995.

[11] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Software Eng. and Methodology*, July 1996.

[12] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Tools for analyzing SCR-style requirements specifications: A formal foundation. Technical report, Naval Research Lab., Wash., DC, 1997. In preparation.

[13] K. Heninger, D. L. Parnas, J. E. Shore, and J. W. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.

[14] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[15] D. N. Hoover and Z. Chen. Tablewise, a decision table tool. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS '95)*, pages 97–108, Gaithersburg, MD, June 1995. IEEE.

[16] J. Kirby. Example NRL/SCR software requirements for an automobile cruise control and monitoring system. Technical Report TR-87-07, Wang Institute of Graduate Studies, 1987.

[17] N. G. Leveson, M. P. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng.*, 20(9), Sept. 1994.

[18] S. P. Miller, Mar. 1997. Personal communication.

[19] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. Softw. Eng.*, 21(2):107–125, Feb. 1995.

[20] D. L. Parnas and J. Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, Oct. 1995.

[21] W. Pugh. A practical algorithm for exact array dependence analysis. *CACM*, 35(8):102–114, Aug. 1992.

[22] M. Shaw. Comparing architectural design styles. *IEEE Software*, 1995.

[23] S. L. Smith and S. L. Gerhart. STATEMATE and cruise control: A case study. In *Proc., COMPSAC*, 1988.

[24] R. M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968. Republished by Dover Publications Inc., 1993.

[25] A. J. van Schouwen. The A-7 requirements model: Reexamination for real-time systems and an application for monitoring systems. Technical Report TR 90-276, Queen's Univ., Kingston, ON, Canada, 1990.

[26] A. J. van Schouwen, Dec. 1996. Personal communication.